Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399

[Haralick 80] R. Haralick, G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[IlogSolver 94] ILOG, 2 Av. Galliéni, F-94253 Gentilly. ILOG SOLVER *Users' Reference Manual*, 1994.

[Mackworth 77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mohr 86] R. Mohr, T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mohr 88] R. Mohr, G. Masini. Good old discrete relaxation. In *Proc. ECAI*, Munich, Germany, 1988.

[Montanari 86] U. Montanari, F. Rossi. *An efficient algorithm for the solution of hierarchical networks of constraints*, volume 291 of *Lecture Notes in Computer Science*, pages 440–457. Springer–Verlag, 1986.

[Prosser 92] P. Prosser, C. Conway, C. Muller. A distributed constraint maintenance system. In *Proc. Les Systèmes Experts et leurs Applications*, Avignon, France, 1992.

[Sabin 94] D. Sabin, E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. ECAI*, Amsterdam, Netherlands, 1994.

[Smith 92] B. Smith. How to solve the zebra problem, or path consistency the easy way. In *Proc. ECAI*, Vienna, Austria, 1992.

[Van Hentenryck 92] P. Van Hentenryck, Y. Deville, C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wallace 93] R. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proc. IJCAI*, Chambéry, France, 1993.

d'explorer les bénéfices de l'utilisation de R|PC au cours de la recherche comme cela est fait avec AC dans l'algorithme MAC décrit dans [Sabin 94].

Développements possibles Les développements possibles de R|PC sont principalement deux généralisations. La première concerne le traitement de contraintes d'arité quelconque en utilisant l'algorithme GAC_4 [Mohr 88], qui est lui même une généralisation d'AC_4 aux contraintes n-aires.

La seconde généralisation concerne la borne que nous avons imposée au nombre maximum de supports d'une affectation au delà duquel on ne s'intéressera pas à déclencher la consistance de chemins. Ici, cette borne était fixée à 1 à cause des bonnes propriétés qui en découlaient. Cependant, il serait probablement intéressant de faire varier cette borne en fonction de la dureté des contraintes.

Références

[Bessière 91] C. Bessière. Arc–consistency in dynamic constraint satisfaction problems. In *Proc. AAAI*, Anaheim, CA, 1991.

[Bessière 94a] C. Bessière, M-O. Cordier. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[Bessière 94b] C. Bessière, J-C. Regin. An arc–consistency algorithm optimal in the number of constraint checks. In *Proc. IEEE International Conference on Tools for Artificial Intelligence (TAI)*, New Orleans, LA, 1994.

[Bessiere 95] C. Bessiere. Une histoire de la cohérence d'arcs (ou comment compter de 1 à 7 en 20 ans). In *Proc.* 5^e *Journées Nationales PRC–GRD Intelligence Artificielle*. Teknea, 1995.

[Dincbas 88] M. Dincbas, P. Van Hentenryck, H. Simonsis, A. Aggoun, T. Graf, F. Berthier. The constraint logic programming language CHIP. In *Proc. FGCS*, 1988.

[Freuder 78] E. Freuder. Synthesizing constraint expression. *Communications of the ACM*, 21(11), 1978.

[Han 88] C. Han, C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

le tableau 1.1 montre que R|PC trouve ses premiers problèmes inconsistants à partir d'une dureté de contraintes de 0.4 alors que, si l'on se limite à l'utilisation d'AC, il faut attendre une dureté de 0.8.

	temps		*#valeurs*		*#échecs*	
dc	AC	RPC	AC	RPC	AC	RPC
0.1	130	135	3	4	0	0
0.2	194	209	4	7	0	0
0.3	248	311	6	12	0	0
0.4	312	488	6	21	0	30
0.5	370	647	7	23	0	87
0.6					0	100
0.7					0	100
0.8					3	100
0.9					10	100
1.0					28	100

Tableau 1.1 : $n = 16, d = 8, sc = 0.5$

	temps		*#valeurs*		*#échecs*	
dc	AC	RPC	AC	RPC	AC	RPC
0.1	119	180	8	23	0	0
0.2	147	293	10	45	0	0
0.3	169	467	12	83	0	39
0.4	204	761	19	121	0	94
0.5					1	100
0.6					4	100
0.7					26	100
0.8					32	100
0.9					62	100
1.0					92	100

Tableau 1.2 : $n = 8, d = 16, sc = 0.8$

Nous avons aussi essayé R|PC sur le problème du Zèbre. La consistance d'arcs est obtenue en 150 ms et retire 31 valeurs du problème. R|PC coûte 210 ms et permet de retirer 37 valeurs, soit 6 valeurs supplémentaires, divisant ainsi par 8 l'espace de recherche obtenu après AC seul.

4.3 Remarques finales

Critiques du recours à l'algorithme AC_4 Certains arguments qui s'opposent à l'utilisation d'AC_4 s'appliquent bien évidemment à l'implantation de R|PC que nous avons décrite ici. Tout d'abord, il y a des critiques importantes à l'encontre de la gourmandise en espace mémoire des structures de données utilisées par AC_4. D'autre part, [Wallace 93] a établi qu'AC_3 est presque toujours plus performant qu'AC_4 pour établir la consistance d'arcs.

Cela est vrai lorsque la consistance d'arcs est utilisée comme une étape de prétraitement mais cela ne l'est plus lorsque la consistance d'arcs est maintenue de façon incrémentale au cours de la résolution d'un problème. Là encore, la raison est que le coût en espace et en temps de construction des structures de données d'AC_4 est bien amorti par leur usage répété au cours de la recherche. Un travail à venir est donc

4.2 Évaluation expérimentale

Afin de vérifier l'efficacité de notre procédure de consistance, nous l'avons expérimentée sur un ensemble de problèmes générés de façon aléatoire. La génération s'effectue de façon classique, à l'aide de 4 paramètres : le nombre de variables n, le cardinal du domaine de chaque variable d, la densité de contraintes dc, exprimée comme une fraction du nombre d'arcs dans un graphe complet à n nœuds et la dureté des contraintes sc, exprimée comme une fraction du nombre de couples du produit cartésien du domaines de deux variables qui ne font pas partie de la relation associée à la contrainte.

Nous avons effectué deux batteries de tests, l'une avec $n = 16$ et $d = 8$ et l'autre avec $n = 8$ et $d = 16$. Pour chacune, nous avons fait varier la densité et la dureté de 0.1 à 1 par pas de 0.1 et pour chaque ensemble de paramètres, nous avons répété l'exécution sur 100 instanciations différentes et reporté la moyenne des résultats obtenus.

Les performances d'AC seul et de R|PC sont comparées sur trois quantités : le temps d'exécution, le nombre de valeurs éliminées et le nombre de problèmes, parmi les 100 construits, qui ont été trouvés inconsistants (c'est à dire qu'un de leur domaine a été réduit à l'ensemble vide). Les tableaux ci-dessous montrent les résultats obtenus pour deux ensembles de paramètres que nous avons trouvés pertinents, c'est à dire que les problèmes engendrés étaient suffisamment contraints pour que la consistance d'arc puisse avoir un effet appréciable, et pas trop contraint pour qu'une inconsistance ne soit détectable immédiatement. Ces tableaux suggèrent deux conclusions sur le comportement de R|PC.

- *Le surcoût entraîné par l'application de* R|PC *est généralement faible par rapport au temps consommé par* AC. Cela est encore plus vrai si on considère ce surcoût relativement au nombre de valeurs éliminées : éliminer une valeur par R|PC coûte entre 2 et 8 fois moins cher que par AC. Cela s'explique évidemment par le fait que la phase la plus coûteuse du processus est la phase d'initialisation des structures faite par AC. L'application de R|PC permet en quelque sorte un meilleur amortissement du travail réalisé pendant cette phase d'initialisation.

- *L'application de* R|PC *offre un pouvoir d'élimination beaucoup plus important qu'*AC *seul.* Sur les résultats obtenus, on voit que R|PC permet de filtrer entre 1.33 et 6.36 fois plus de valeurs qu'AC. La procédure R|PC permet donc de détecter des problèmes inconsistants beaucoup plus tôt qu'AC. Par exemple,

– Cet arc peut être le seul à pouvoir composer un chemin entre la variable i et une troisième variable k telle que a ne possède qu'un seul support sur k. Ce cas est traité par les lignes 10 à 12.

Lorsque *elimine* termine, on invoque alors la procédure *verifie* avec la liste $list_{PC}$ qui vient d'être déterminée. Pour chaque triplet $(\langle j, b\rangle, i, k)$ de cette liste, on détermine dans S_{jb} la valeur a qui constitue le seul support de b sur i (ligne 4).

Puis, étant donnée la variable k liée à la fois à i et j, on vérifie qu'il existe un chemin de b à a passant par k. S'il n'y en a pas, on sait alors que la valeur b peut être éliminée du domaine de j. On décrémente alors son compteur et on propage sa suppression dans le graphe en relançant la procédure *elimine*.

Notons que les triplets qui deviennent pertinents pour *verifie* à la suite des applications successives de *elimine* sont ajoutées incrémentalement à $list_{PC}$ par *elimine* elle même.

Comme pour la consistance d'arcs, la terminaison de R|PC est garantie par la diminution monotone du nombre de couples examinables et par la diminution de la liste $list_{PC}$ assurée par la ligne 2 de *verifie*.

4.1.2 Complexité

Concernant la complexité en espace et par rapport à AC_4, notre algorithme introduit une liste supplémentaire, $list_{PC}$. Cette liste atteint sa taille maximale lorsque, pour toutes les variables du problème, chaque affectation n'a qu'un seul support par rapport à chacune des contraintes. Dans ce cas, la taille de la liste est en $O(ned)$. La complexité en espace d'AC_4 étant en $O(ed^2)$, la borne supérieure de la complexité en espace de R|PC est en $O(ed(n+d))$.

On s'intéresse maintenant à la complexité en temps. Les modifications que nous avons apportées aux procédures *initialise* et *elimine* ne changent pas la complexité d'AC_4. En ce qui concerne la procédure *verifie*, la boucle de la ligne 1 est exécutée au plus ned fois. Bien entendu, le coût des exécutions subséquentes de *elimine*, appelée à la ligne 7, est couvert par le coût calculé pour la consistance d'arcs. La complexité totale de *verifie* est donc en $O(ned)$.

Comme pour la complexité en espace, on abouti à une borne supérieure en $O(ed(d+n))$. Pour mémoire, nous rappelons que la complexité en temps du meilleur algorithme de consistance de chemins, PC_3 [Han 88], est en $O(n^3d^3)$.

```
{étape 1 : construction des structures de donnée}

procedure initialise(var list_AC , var list_PC );
1  pour tout (i, j) tel que C_ij ∈ C faire
2       pour tout a ∈ D_i faire
3            total ← 0;
4            pour tout b ∈ D_j faire
5                 si (a, b) ∈ R_ij alors
6                    total ← total + 1;
7                    S_ia ← S_ia ∪ {⟨j, b⟩};
8            compteur[(i, j), a] ← total;
9            si compteur[(i, j), a] = 0 alors
10              list_AC ← list_AC ∪ {⟨i, a⟩}; D_i ← D_i \ {a}
11              sinon si compteur[(i, j), a] = 1 alors
12                       pour tout k ∈ V tel que C_ik et C_kj ∈ C faire
13                            list_PC ← list_PC ∪ {(⟨i, a⟩, j, k)}.

{étape 2 : élimination des valeurs inconsistantes}

procedure elimine(list_AC , var list_PC );
1  tantque list_AC ≠ ∅ faire
2      choisir et retirer ⟨j, b⟩ de list_AC ;
3      pour tout (i, a) ∈ S_jb faire
4           compteur[(i, j), a] ← compteur[(i, j), a] − 1;
5           si compteur[(i, j), a] = 0 et a ∈ D_i alors
6              list_AC ← list_AC ∪ {⟨i, a⟩}; D_i ← D_i \ {a}
7           sinon si compteur[(i, j), a] = 1 alors
8                    pour tout k ∈ V tel que C_ik et C_kj ∈ C faire
9                         list_PC ← list_PC ∪ {(⟨i, a⟩, j, k)}
10                sinon pour tout k ∈ V tel que C_ik et C_kj ∈ C faire
11                         si compteur[(i, k), a] = 1 alors
12                            list_PC ← list_PC ∪ {(⟨i, a⟩, k, j)}.

{étape 3 : vérification de l'existence de chemins}

procedure verifie(var list_PC );
1  tantque list_PC ≠ ∅ faire
2      choisir et retirer (⟨j, b⟩, i, k) de list_PC ;
3      si b ∈ D_j alors
4         soit {⟨i, a⟩} = {⟨i, x⟩ ∈ S_jb | x ∈ D_i};
5         si {⟨k, c⟩ ∈ S_ia | c ∈ D_k} ∩ {⟨k, c⟩ ∈ S_jb | c ∈ D_k} = ∅ alors
6            compteur[(j, i), b] ← 0; D_j ← D_j \ {b};
7            elimine({⟨j, b⟩}, list_PC ).
```

FIG. 6 - *calcul de la consistance de chemins limitée*

A l'instar de la consistance d'arcs, on dira qu'un problème est RP-consistant ssi toutes ses variables sont RP-consistantes. Par définition, un problème RP-consistant est aussi arc-consistant (la réciproque est fausse). Un problème chemin-consistant est évidemment RP-consistant (et ici encore, la réciproque est fausse).

4.1 Algorithme

Pour établir la RP-consistance, il faut à la fois établir la consistance d'arcs et sélectionner l'ensemble des couples d'affectations à vérifier. Pour réaliser cette dernière opération, il faut compter le nombre de supports de toutes les affectations possibles par rapport à l'ensemble des contraintes. Or, cette comptabilité est précisémment la clé de voûte de l'algorithme AC_4 [Mohr 86], ce qui en fait un choix naturel pour le développement de R|PC.

4.1.1 Présentation détaillée

La figure 6 montre les trois procédures utilisées pour établir la RP-consistance. Les procédures *initialiser* et *elimine* sont directement dérivées de l'algorithme AC_4. Comme dans l'algorithme original, *initialiser* construit deux structures : S_{ia} qui enregistre l'ensemble des affectations qui sont supportés par la valeur a pour la variable i et $compteur[(i,j),a]$ qui comptabilise le nombre de supports sur j de l'affectation $\langle i,a\rangle$.

Les différences de *initialise* et *elimine* par rapport à leurs versions initiales sont l'addition respective des lignes 11 à 13 et 7 à 12.

Pour *initialise*, ces trois lignes permettent, lorsqu'on vient d'initialiser le compteur de supports d'une affectation de vérifier si l'affectation en question ne possède qu'un seul support. Si c'est le cas, on stocke cette information dans la liste $list_{PC}$ sous la forme d'un triplet qui décrit l'affectation supposée fragile, la variable sur lequel persiste le dernier support, et la troisième variable par laquelle on cherchera l'existence d'un chemin.

Lorsqu'on atteint la ligne 7 de la procédure *elimine*, c'est qu'on vient de retirer l'arc $(\langle i,a\rangle,\langle j,b\rangle)$ et que ce retrait n'a pas provoqué l'élimination de la valeur a pour i. Deux cas se présentent alors :

- Cet arc peut être le dernier qui soutient la valeur a sur i vis-à-vis de j. On est alors ramené au cas traité par les lignes 11 à 13 de *initialise*.

cessivement la valeur e pour k et la valeur d pour j. Le problème résultant (qui, incidemment, devient globalement consistant) est montré sur la figure 5.2.

On voit donc qu'en ne s'intéressant qu'à un nombre limité de couples correctement choisis, il est possible d'opérer un filtrage de domaines plus puissant que celui obtenu par la consistance d'arcs. De par son caractère incomplet, R|PC s'affranchit des inconvénients de la consistance de chemins, à savoir que :

- la complexité de la procédure est réduite puisque le nombre de couples de valeurs à considérer est fortement limité.
- aucune représentation particulière des contraintes n'est nécessaire puisqu'on ne retire pas de couples d'affectation.
- aucune contrainte supplémentaire n'est créée puisqu'encore une fois, on ne retire pas de couples.

Avant de s'intéresser plus en détail à la procédure R|PC, on donne une définition formelle du niveau de consistance partielle qu'elle permet d'établir. Tout d'abord, les conditions qui caractérisent la consistance de chemin peuvent être définies formellement de la façon suivante :

- Un couple de variables $\{i, j\}$ est dit chemin-consistant ssi :
  - i et j sont arc-consistantes,
  - pour tout $(a, b) \in D_i \times D_j$
    $k \in \mathcal{V}$ tel que C_{ik} et $C_{kj} \in \mathcal{C}$,
    il existe $c \in D_k$ tel que $(a, c) \in R_{ik}$ et $(c, b) \in R_{kj}$.

En prenant appui sur cette définition et celle de la consistance d'arcs donnée dans la section 2, on peut alors formuler la définition suivante :

- Une variable i est dite RP-consistante ssi :
  - i est arc-consistante,
  - pour tout $a \in D_i$,
    $j \in \mathcal{V}$ telle que $C_{ij} \in \mathcal{C}$ et $\exists! b \in D_j, (a, b) \in R_{ij}$,
    $k \in \mathcal{V}$ telle que C_{ik} et $C_{kj} \in \mathcal{C}$,
    il existe $c \in D_k$ tel que $(a, c) \in R_{ik}$ et $(c, b) \in R_{kj}$